Efficient computation of solution space and conflicts detection for linear systems

Alexandre BORTHOMIEU
University of Grenoble
INRIA - STEEP Team
Supervised by Jean-Yves Courtonne
alexandre.borthomieu@etu.univ-grenoble-alpes.fr

ABSTRACT

In this paper, we analyze a system of linear inequations $l \leqslant A.x \leqslant u$, used for the purpose of Material Flow Analysis, with three different but complementary goals : (i) given some known variables x_i, efficiently compute the solution space of unknown variables, (ii) if the set of constraints is infeasible, efficiently identify the conflicts, (iii) efficiently classify variables to determine whereas they are redundant, just measured, determinable or non-determinable. In each case we compare the efficiency of different algorithms or languages.

1. INTRODUCTION

In a context of global environmental changes which questions our modes of production and consumption, the STEEP team of INRIA aims at developing decision-making tools to promote ecological transition at local (sub-national) levels. One of these tools, called Supply Chain Material Flow Analysis (MFA) tries to provide a consistent image of a given chain (e.g., the wood supply chain) in order to stress the potential competition between different types of uses (e.g., energy vs. construction) as well as the material dependency on imports from the rest of the world. In practice, flows are the variables of the problem (some are measured, some are unknown) and the constraints are given by the law of mass conservation and by other rules linking variables (e.g., yields of transformation). The MFA follows the following steps :

1. Reconcile the data with constraint optimization, i.e., minimize the distance between measured flows (inputs) and model results (outputs) given the set of constraints. Reconciliation is necessary because the data comes from various sources which are generally inconsistent. Constraint optimization moreover "fills the gap" for missing flows. However, it provides a unique solution for each variable even when the solution is not unique: for the so-called "free" variables, other values would also respect constraints and lead to the same value of the objective function. It is thus important to identify them.

2. For this purpose, variables are classified based on the set of constraints and on the measured variables in order to determine which ones are determinable (not measured but deductible), free (not measured and not deductible), redundant (measured and staying determinable if the measure is dropped), just measured (measured and becoming free if the measure is dropped).

3. Knowing which variables are "free" and knowing the unique solution for other variables, an interval (solution space) is computed for each free variable.

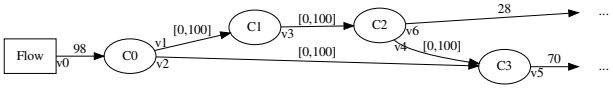
STEEP already developed a python program to conduct these 3 steps. However it proves very inefficient for large problems (about 20,000 variables). Another issue is that in practice, when one builds a large supply chain model, it often occurs that the reconciliation is infeasible because of conflicting constraints, and the manual identification of these constraints is a tedious job. The purpose of our work was therefore to make improvements on these two fronts.

In the following sections, we start by presenting the problem of interval reduction (step 3 above) before turning to conflicts identification. We finish with ongoing and future work, particularly, in relationship with variable classification.

1.1 Our examples

To illustrate our algorithms, we will take two examples: the first one to computing the solution space of "free" variables and the second one for the conflicts detection algorithm. Both will take the form of a simple flow-graph as a reminder of our context.

Figure 1: Without conflicts

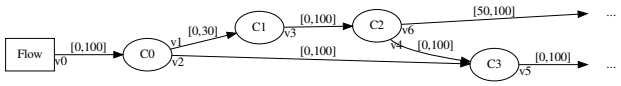


In this first example, we suppose that we have measured or determined some variables (v_0, v_6,v_5). Others remain missing. We will apply our algorithm of Intervals Reduction (IR) with the hypothesis that every variable should be between 0 and 100.

We will use this second example with our algorithm of Conflicts Detection (CD). Indeed, in this example, we have added two constraints on v_1 (C_4) and v_6 (C_5) in order to create an unfeasible system. Those constraints can not be represented by flows because they constrain only one variable. So, we can observe them by looking at the intervals of v_1 ($[0, 30]$) and v_6 ($[50, 100]$).

2. INTERVALS REDUCTION

Figure 2: With conflicts



To compute new intervals, we work on a linear system. We try to determine for each variables a maximum and a minimum to surround the solution space. Thus, let say we have N constraints $c_0...c_{N-1}$ and M variables $v_0...v_{M-1}$ such as :

$$\forall i \in [|0..N-1|],\ c_i : l_i \leqslant \sum_{k=0}^{M-1} \alpha_{i,k} v_k \leqslant u_i \tag{1}$$

with l_i and u_i the bounds of each constraint and $\alpha_{i,k}$ the coefficient of the variable k in the constraint i.

In our context, we said that some variables would be fixed as far as we know their values. Thus, let F be the number of variables known. All variables that are not "free" are known variables (they are uniquely determined). Let L be the number of unknown variables such as $M = F + L$. We can now decompose our first equation (1) :

$$\forall i \in [|0..N-1|],\ c_i : l_i \leqslant \sum_{f=0}^{F-1} \alpha_{i,f} v_f + \sum_{l=0}^{L-1} \alpha_{i,l} v_l \leqslant u_i \tag{2}$$

Our goal here is to compute the maximum and minimum values that each v_l can achieve. To do so, a few algorithms already exist.

2.1 Similar work

An interesting algorithm can be found in the literature. Indeed, this problem appears similar to the optimization problem of the *Simplex*[1]. The *Simplex* method consists in approaching the optimal solution of an objective function in a problem under duress. Many *Simplex* algorithms have been developed and are extremely efficient. Thus, in their paper, Puranik and Sahinidis [2] develop the idea that if we run two *Simplex* programs for each variable (one to compute the maximum and one for the minimum), we are able to get the interval of each unknown variable. Therefore, we have to run $2 * L$ *Simplex* to solve our system. In term of complexity, running a such powerful tool two times on variables sub-problem could take some time. But we will detail the efficiency of that algorithm in the part 2.4.

When I joined the STEEP Team, one algorithm was already developed and was particularly long. This algorithm uses the property of each node in the flow-graph. For each constraint, if we want to isolate one unknown variable of index k to compute his maximum or minimum, we get :

Let $k \in [|0..L-1|],\ \forall i \in [|0..N-1|]$,

$$c_i \begin{cases} l_i \ - \sum_{f=0}^{F-1} \alpha_{i,f}\ v_f - \sum_{l=0,l\neq k}^{L-1} \alpha_{i,l}\ v_l \leqslant \boldsymbol{\alpha_{i,k} v_k} \\ \boldsymbol{\alpha_{i,k} v_k} \leqslant u_i \ - \sum_{f=0}^{F-1} \alpha_{i,f}\ v_f - \sum_{l=0,l\neq k}^{L-1} \alpha_{i,l}\ v_l \end{cases} \tag{3}$$

For each v_l, we can only get an interval. In order to calculate the value of v_k, we need to use the maximum or the minimum of each variable v_l which depends on the coefficient $\alpha_{i,k}$. Thus, we generalize this with four cases as follow. We place ourselves in a context of one constraint with two unknown variables to simplify the inequations. With the graph, on a node, we observe in-flow and out-flow. These flows are represented by the sign of the coefficient α_X. Furthermore, when a flow v_l (unknown variable) has the same sign than v_k (unknown variable for which we wish to tighten his interval), we use $v_{l_{min}}$ (resp. $v_{l_{max}}$) to compute $v_{k_{max}}$ (resp. $v_{k_{min}}$). On the other hand, if coefficients have a different sign, we use $v_{l_{min}}$ (resp. $v_{l_{max}}$) to compute $v_{k_{min}}$ (resp. $v_{k_{max}}$). In this formalization, the coefficients α_X are positive.

- Case 1 : $l \leqslant \alpha_k v_k + \alpha_l v_l \leqslant u$

$$\begin{cases} v_{k_{max}} \leqslant \frac{1}{\alpha_k}(u - \alpha_l v_{l_{min}}) \\ v_{k_{min}} \geqslant \frac{1}{\alpha_k}(l - \alpha_l v_{l_{max}}) \end{cases} \tag{4}$$

- Case 2 : $l \leqslant \alpha_k v_k \boldsymbol{-\alpha_l}\ v_l \leqslant u$

$$\begin{cases} v_{k_{max}} \leqslant \frac{1}{\alpha_k}(u + \alpha_l v_{l_{max}}) \\ v_{k_{min}} \geqslant \frac{1}{\alpha_k}(l + \alpha_l v_{l_{min}}) \end{cases} \tag{5}$$

- Case 3 : $l \leqslant \boldsymbol{-\alpha_k} v_k \boldsymbol{-\alpha_l}\ v_l \leqslant u$

$$\begin{cases} v_{k_{max}} \leqslant \frac{1}{-\alpha_k}(l + \alpha_l v_{l_{min}}) \\ v_{k_{min}} \geqslant \frac{1}{-\alpha_k}(u + \alpha_l v_{l_{max}}) \end{cases} \tag{6}$$

- Case 4 : $l \leqslant \boldsymbol{-\alpha_k} v_k + \alpha_l v_l \leqslant u$

$$\begin{cases} v_{k_{max}} \leqslant \frac{1}{-\alpha_k}(l - \alpha_l v_{l_{min}}) \\ v_{k_{min}} \geqslant \frac{1}{-\alpha_k}(u - \alpha_l v_{l_{max}}) \end{cases} \tag{7}$$

In this algorithm, we loop over the unknown variables to compute a maximum and a minimum for each constraint with those properties. We repeat this operation while the intervals are being modified.

2.2 Our Algorithm

In order to find a new algorithm, we manually tried to find the intervals for our small example, without looking at existing algorithms. Then, we tried to automate our method in an algorithm. Finally, we compared other algorithms with our own method. We found out that our algorithm was really close to the STEEP Team's one. Actually, the differences came from the order of the for-loop. We loop first over the constraints and then over unknown variables to compute the intervals (instead of doing the opposite). Also, a list of constraints to check was added to increase efficiency. When a variable is modified, we put in this list all constraints in which our variable occurs. We pop out those constraints once they are checked. Thanks to that, we do not need to loop over every constraints each time a variable is modified.

[1]https://en.wikipedia.org/wiki/Simplex_algorithm

Algorithm 1: Intervals Reduction

```
intervals reduction (M, I, V, u, l) :
  Input  : M a matrix of coefficient, I a list of the
           intervals, V a list of the unknown variables, u
           and l the lists of the boundaries of each
           constraint
  Output: a list of the reduced intervals
C ← all the constraints
while I is modified do
    for each constraint c_i in C do
        Drop c_i
        for each unknown variable v_k in c_i do
            Compute v_{k_max} and v_{k_min} (With the four
             cases (4) - (7))
            if v_{k_max} ⩽ upper-bound v_k then
                Modify the upper-bound v_k
            end
            if v_{k_min} ⩾ lower-bound v_k then
                Modify the lower-bound v_k
            end
            if v_k 's interval is modified then
                Add to C every c_j in which v_k occurs
            end
        end
    end
end
return I
```

Here, before this algorithm we use a pre-processing to compute the new boundaries (i.e. fixed values are being subtracted from the l_i, u_i bounds) for each constraint as follow :

$$\forall i \in [|0..N-1|], \begin{cases} l_i \leftarrow l_i - \sum_{f=0}^{F-1} \alpha_{i,f} v_f \\ u_i \leftarrow u_i - \sum_{f=0}^{F-1} \alpha_{i,f} v_f \end{cases} \quad (8)$$

Furthermore, our algorithm stops because either we reach a max iterations that we fixed or the intervals are unmodified which end the while loop.

With this method, we approach the optimal intervals with more efficiency. Indeed, by computing in each constrain all the unknown variables, we obtain a better restriction on the other variables. That is because the intervals of more variables are reduced in one loop. And the more there are restrictions, the more the intervals are quickly tighten to their optimal values.

Moreover, we can apply a few optimization methods easily (such as parallelism) as the form of the two for-loop are simple (thus easily parallelisable). This tool can drastically increase the speed of our algorithm. To confirm such predicate, we have to create tests that will give us the performance data for our analysis. But first, let us illustrate our algorithm with our example.

2.3 Application to our example

With our example (Figure 3), the system will be stored in a matrix as follow :

$$\begin{array}{c} \\ C_0 \\ C_1 \\ C_2 \\ C_3 \end{array} \begin{array}{c} \begin{array}{ccccccc} v_0 & \boldsymbol{v_1} & \boldsymbol{v_2} & \boldsymbol{v_3} & \boldsymbol{v_4} & v_5 & v_6 \end{array} \\ \begin{pmatrix} 1 & -1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & -1 & 0 & -1 \\ 0 & 0 & 1 & 0 & 1 & -1 & 0 \end{pmatrix} \end{array}$$

v_1, v_2, v_3 and v_4 are our unknown variables here. Once this matrix is constructed and the initialization of our intervals is done, we start the loop over the constraints. C_0 will give us new boundaries on v_1 and v_2 such as :

$$\begin{cases} v_{1_{max}} \leqslant \frac{1}{-1}(-98+0) \implies v_{1_{max}} \leqslant 98 \\ v_{1_{min}} \geqslant \frac{1}{-1}(-98+100) \implies v_{1_{min}} \geqslant -2 \end{cases} \quad (9)$$

We keep the upper-bound only because we want our interval tightened as possible. Thus, v_1 is in $[0, 98]$.

$$\begin{cases} v_{2_{max}} \leqslant \frac{1}{-1}(-98+0) \implies v_{2_{max}} \leqslant 98 \\ v_{2_{min}} \geqslant \frac{1}{-1}(-98+98) \implies v_{2_{min}} \geqslant 0 \end{cases} \quad (10)$$

We continue the loop over the constraints while there are still some modifications. When the algorithm ends, we get the following result :

Figure 3: Result Intervals Reduction

2.4 Efficiency

Our tests work as follow : for each algorithm we run four examples of different sizes and store the results. First, we want to warm-up our CPU. Thus, ten call to our algorithm are launched without any measurements of the execution time. Then, we run one hundred instances on our example and we compute the average of those launches. At last, we do this operation ten times and as a result, we get the average of these ten averages[2]. As for now, the final execution time is quite representative of the efficiency of this algorithm.

In this table, each average (computed like mentioned above) is stored depending on the size of the example and the algorithm employed. Each number is in milliseconds (except with the huge example).

	Small[1]	Medium[2]	Medium[3]	Huge[4]
Simplex	113	/	/	/
STEEP	0.303	105.30	438	∼1h
STEEP (numba)	0.04	9.04	77	∼11min
Our Algo	0.02	2.00	6	∼1s
Our Algo (numba)	0.03	0.75	2	76ms
Our Algo (pybind)	0.004	0.12	0.4	31ms

[1] around 10 constraints and 4 unknown variables
[2] around 300 constraints and 150 unknown variables
[3] around 800 constraints and 400 unknown variables
[4] around 18 000 constraints and 16 500 unknown variables

We developed in python language all those algorithms. However, to use parallelism, we use two kind of tools. The

[2]Theorem Central Limit

first one we thought of was the *numba* tool. It is a python library which permits to "translate python function to optimized machine code at runtime"[3]. Thanks to that, we can approach the speeds of a C program. There are many constraints on types of all the arguments, pre-processing was necessary to redefine our parameters. The second tool is called *pybind*. It consists in writing a C++ code of our algorithm with some links to python types. Then compiling it in a python library with many optimization processes. Finally, we can use this library in a python program to compute our intervals faster thanks to parallelism and the C like code.

Concerning the *Simplex* program, we did not succeed to run it on bigger examples and we are trying to understand why. However, this program takes a long time on a very small example compare to the other ones.

3. CONFLICTS DETECTION

The conflict detection is used to identify conflicting constraints in order to modify them and make the problem feasible. A variable shouldn't be getting two disjointed intervals with two constraints. In this case, our previous algorithm will notify us. But once our algorithm detect a conflict, we need to know which constraints are involved in this conflict. Indeed, the source of a conflict can be many loop before it is notified. A variable can be modify several times before encounters a constraint in which it will get a disjointed interval. Moreover, the conflict can come from another variable that affects this one. So, how can we determine which constraints should be verify ?

3.1 Our Algorithm

Our first idea was to store each modification of a variable when IR is running and try to reverse the path that led to a conflict. However, it was extremely difficult to analyze each possibilities to compute the correct one and the complexity of that idea in term of time and space wasn't satisfactory. Then, we discover the work of J.W. Chinneck [1] that Y. Puranik and N.V. Sahinidis [2] took over. In his article, J.W. Chinneck presents two algorithms to find a subset of constraint which is infeasible in a linear program. The first one called the *Deletion Filter*, consists in deleting constraints one by one until the linear program becomes feasible. Once we reach this point, we put back the last constraint dropped in the set and we keep deleting the other constraints. Here is their algorithm (Algorithm 2).

The second algorithm called *Additive Filter* works on the same idea. We add constraints to a set T until this set becomes infeasible. Then we add the last constraint to the I_S infeasible set and start over with $T = I_S$. Here is their algorithm (Algorithm 3).

To detect if a set is infeasible, we use our *Intervals Reduction* algorithm. When a variable gets two disjointed intervals, we have an infeasible set :

$$v_{k_{max}} \leqslant \text{lower-bound } v_k \text{ or } v_{k_{min}} \geqslant \text{upper-bound } v_k \implies \text{infeasible set} \quad (11)$$

Both algorithms end with one irreducible subset of infeasible constraints. Sometimes, our linear system could have

[3]according to https://numba.pydata.org/

Algorithm 2: Deletion Filter

```
1  deletion filter (I) :
   Input  : I a set of infeasible constraints
   Output: an irreducible subset of infeasible constraints
2  for each constraint c_i in I do
3  |  Temporarily drop c_i
4  |  if the set become feasible then
5  |  |  Return c_i to the set
6  |  end
7  |  else
8  |  |  Drop the constraint permanently
9  |  end
10 end
11 return I
```

Algorithm 3: Additive Filter

```
1  additive filter (I) :
   Input  : I a set of infeasible constraints
   Output: an irreducible subset of infeasible constraints
2  T = I_S = ∅
3  while I_S is feasible do
4  |  T ← I_S
5  |  for each constraint c_i in I do
6  |  |  T ∪ {c_i}
7  |  |  if T is infeasible then
8  |  |  |  I_S ← I_S ∪ {c_i}
9  |  |  |  GOTO while
10 |  |  end
11 |  end
12 end
13 return I_S
```

more than one conflict. To include this possibility, we add a while loop around both algorithms which loop if the set I is still infeasible without the irreducible subset just found. With this modification we are able to collect all the irreducible subsets of our linear system.

Now that we have two algorithms that compute irreducible infeasible subsets of constraints, we want to imagine a new one that could give us the same result faster. Indeed, in both algorithms, constraints are added or deleted one by one and in a huge linear system that takes time. Y.Puranik and N.V. Sahinidis [2] mentioned in their paper that those algorithms can be modify by adding or deleting constraints by packs.

With this idea, we thought about a new algorithm. This algorithm consists in apply the *Additive Filter* by adding constraints by packs. Then we have I_S which is not an irreducible infeasible subset but a smaller infeasible subset. We apply the *Deletion Filter* algorithm to this subset. We finally get one irreducible subset. We repeat this until I becomes feasible. To sum up, we add constraints by pack and we drop them one by one. We called it the *Hybrid Filter*.

3.2 Our example

We will process our second example 2 with the algorithm *Deletion Filter*.

First, C_0 is dropped from I. The set remains infeasible so this constraint is dropped permanently. Now, C_1 is processed, when this constraint is dropped, the subset becomes feasible. That means this C_1 is a part of the problem. So, this constraint will remain in the subset. Next, C_2 will be equivalent to C_1 and C_3 will be permanently removed because the subset remains infeasible without them. This algorithm will now process C_4 and C_5 and find out that they are also a source to our problem. Eventually, this algorithm will give us the following subset : $[C_1, C_2, C_4, C_5]$.

3.3 Efficiency

The efficiency (measured with the same process details in the section 2.4) of those three algorithms is stored in this table. Like said in the section 3.1 with the expression (11), we use our IR algorithm. In order to reduce the execution time, we choose the IR algorithm with the *numba* structure which is easier to call.

	Small[1]	Medium[2]	Medium[3]	Huge[4]
Deletion Filter	0.89ms	1.2s	10s	+1h
Additive Filter	0.93ms	1.2s	9.5s	+1h
Hybrid Filter	1.60ms	2.3s	18s	+1h

[1] around 10 constraints and 4 unknown variables
[2] around 300 constraints and 150 unknown variables
[3] around 800 constraints and 400 unknown variables
[4] around 18 000 constraints and 16 500 unknown variables

For those three algorithms, an huge example takes too much time. We stop the execution once it exceed one hour. We notice that our *Hybrid Filter* algorithm is not faster like expected. But, it was predictable. Indeed, in this algorithm, to add a pack of constraints, we are forced by our structure to add them one by one. Thus, the execution time is far more high than expected. The *Deletion Filter* and the *Additive Filter* algorithms show that on particularly large linear system, an algorithm more efficient is needed to find infeasible subsets.

4. ONGOING AND FUTURE WORK

We recently realize that our IR algorithm is not totally accurate. Indeed, in a particularly case, the intervals computed are incorrect. When there are dependencies between unknown variables and constraints, some intervals can not be computed correctly. Here is an example, let say we have a constraint such as $C_0 : a = b - c$ and a second one such as $C_1 : x + b = y + c$ where a, b and c are unknown and x and y known. With some inequalities, we have :

$$\begin{cases} 0 \leqslant a - b + c \leqslant 0 \\ y - x \leqslant b - c \leqslant y - x \end{cases} \tag{12}$$

If some constraints look like those in our linear system, some intervals could be computed with infinite value, which is incorrect. Indeed, in this example, our algorithm will compute a with infinite bounds instead of $y - x$. One idea is to modify our matrix to a special form called *RREF* (Reduced Row Echelon Form) which is a form that we use in our variable's classification algorithm [3]. So, this tool can be easily used for our matrix here. However, this tool also takes time on large matrix. Our current and future work is to transform our current code of *RREF* into a *pybind* structure and then apply it to our matrix. Then, we will verify our assumption with tests and demonstrations. If this idea does not work, we will have to find other methods to have a IR algorithm applicable to any example.

Secondly, like we said in our 3.3 section, the *Hybrid Filter* algorithm add constraints one by one because of our data's type. We could think of a new data structure which will allow us to add constraints by pack and truly exploit our idea. Potentially, we could think of new algorithm with a kind of dichotomy to approach infeasible subsets faster.

5. CONCLUSION

Both efficiency tables showed in this paper help us to understand where our theoretical ideas were inaccurate and where we could still gain execution time. This paper gives a new tool to the tightening intervals problem with improvements still possible. This algorithm can now be added to the STEEP project in order to increase their time efficiency on their MFA tool. Indeed, we divide here our execution time by thirty six hundred without any parallelism. This work also gives new ideas to develop a high efficiency on conflicts detection with a large number of constraints. Despite our wrong expectations on our experimental measurements with the *Hybrid Filter*, we know how to transform our data structure to increase the performance on this topic. Moreover, the classification method should now be easily improved by the ongoing work on the *RREF* algorithm.

6. REFERENCES

[1] J. W. Chinneck. Finding a useful subset of constraints for analysis in an infeasible linear program. *INFORMS Journal on Computing*, 9(2):164–174, 1997.

[2] Y. Puranik and N. V. Sahinidis. Deletion presolve for accelerating infeasibility diagnosis in optimization models. *INFORMS Journal on Computing*, 29(4):754–766, 2017.

[3] V. Ververka and F. Madron. Material and energy balancing in the process industries, 1997.